Universidad Centroamericana
José Simeón Cañas

Análisis de algoritmos

Evaluacion:
Taller 2

Nombre de los integrantes:
Elías Ernesto Zelaya Arias 00146322
Carlos Alejandro Domínguez Renderos 00116822
Andres Felipe Cardona Duarte 00037820

San Salvador 12 de octubre de 2024.

Etapa IV: Análisis del código:

```cpp
#ifndef Employee_H
#define Employee_H

#include <string>
#include <iostream>
#include <fstream>
#include <sstream>

using namespace std;

struct Employee {
    string name;
    float salary;
};
```

O(1)

1. En este bloque de código, el orden de magnitud es O(1), ya que solo se está declarando una estructura, y utilizando librerias, todo esto se considera como operaciones primitivas, por lo que el orden de magnitud es constante

```
Employee* load_data() {                                                  C1
    ifstream infile("MOCK_DATA.txt");                                    C2
    string line;                                                         C3
    int num_employee = 0;                                                C4
    Employee* employees = new Employee[1500];                            C5

    if (infile.is_open()) {                                              C6
        while (getline(infile, line)) {                                  C7(n+1)
            stringstream ss(line);                                       C8(n)
            string name;                                                 C9(n)
            string salaryStr;                                            C10(n)

            if (getline(ss, name, ',') && getline(ss, salaryStr)) {      C11(n)*max(0, 1)
                double salary = stod(salaryStr);

                employees[num_employee].name = name;                     C12(n)
                employees[num_employee].salary = salary;
                num_employee++;
            }
        }

        infile.close();                                                  C13
    } else {
        cout << "Failed to open the file." << endl;                      C14
    }

    return employees;                                                    C15
}
```

C1+C2+C3+C4+C5+C6+C7(n+1)+C8(n)+C9(n)+C10(n)+C11(n)*max(0, 1)+C12(n)+C13+C14+C15

C1+C2+C3+C4+C5+C6+C7(n)+C7+C8(n)+C9(n)+C10(n)+C11(n)+C12(n)+C13+C14+C15

$$\underbrace{C1+C2+C3+C4+C5+C6+C7+C13+C14+C15}_{A} + \underbrace{C7(n)+C8(n)+C9(n)+C10(n)+C11(n)+C12(n)}_{B(n)}$$

T(n) = A + B(n)

T(n) = O(1) + O(n)

T(n) = O(n)

2. Para la funcion load_data, se analizar parte por parte, las primeras líneas antes del while son solo declaraciones de variables y el caso de las cabeceras del if que es constantes y se multiplica por MAX(0, 1), ya que se elige el peor de los casos, y en este caso es que la condicion se cumpla, por lo que se multiplica por uno y eso da la misma constante, por lo tanto, su tiempo de ejecución es constante.

   al llegar al primer while, se analiza la cabecera y se ejecuta de manera lineal, por el hecho de que recorre cada línea del fichero, hasta llegar al final de esta, por lo que el tiempo de ejecucion de la cabecer vendria siendo n+1 por el hecho de que la cabecera se ejecuta una vez más que el cuerpo, y el cuerpo de dicho bucle sería de n.

Dentro del while se encuentra un if, por lo que volvemos a utilizar MAX y el valor constante se multiplica nuevamente por uno ya que se asume que la condicion se cumple, ya luego todo lo que se encuentra dentro del while tiene tiempo de ejecucion n y lo que se encuentra despues del while es constante, por lo tanto, el tiempo de ejecucion de toda la funcion load_data es O(n)

```cpp
#ifndef MINHEAP_H
#define MINHEAP_H

#include <string>
#include <iostream>
#include "employee.h"

using namespace std;

struct minHeap{
    Employee* arr;

    int size;

    int capacity;

};
```

O(1)

3. En este bloque de código, su tiempo de complejidad es O(1), ya que solo se incluyen las librerias y declaracion de la estructura para los heaps

```
int parent(int i) {
 return (i - 1) / 2;
}

int left(int i) {
 return 2 * i + 1;
}

int right(int i) {
 return 2 * i + 2;
}

void swap(Employee* a, Employee* b) {
 Employee temp = *a;
 *a = *b;
 *b = temp;
 }

 minHeap* init_MinHeap(int capacity) {
 minHeap* heap = new minHeap;
 heap->size = 0;
 heap->capacity = capacity;
 heap->arr = new Employee[capacity];
 return heap;
}
```

O(1)

4. En esta parte del código, el tiempo de complejidad para las 5 funciones es constante, debido a que en todas solo se realizan operaciones aritméticas, asignacion de valores a las variables y paso de parametros a las funciones, solo en el caso de la funcion init_MinHeap, en la que se crea un arreglo con una variable capacity, pero desde el lugar donde se llama esta se le envía un dato constante, por lo tanto sigue siendo constante, por lo tanto, el tiempo de ejecucion de todo ese bloque de funciones es O(1)

```
minHeap* insert(minHeap* heap, Employee employee) {
    if (heap->size == heap->capacity) {                                               C1*max(0, 1)
        cout << "Overflow: Could not insert key\n";                                   C2
        return heap;                                                                  C3
    }

    heap->size++;                                                                     C4
    heap->arr[heap->size - 1] = employee;                                             C5

    int key = heap->size - 1;                                                         C6

    while(key > 0 && heap->arr[parent(key)].salary > heap->arr[key].salary) {         C7(lg(n)+1)
        swap(&heap->arr[parent(key)], &heap->arr[key]);                               C8(lg(n))(O(1))

        key = parent(key);                                                            C9(lg(n))(O(1))
    }
    return heap;                                                                      C10
}
```

$$C1*\max(0, 1)+C2+C3+C4+C5+C6+C7(\lg(n)+1)+C8(\lg(n)(O(1))+C9(\lg(n)(O(1))+C10$$

$$C1+C2+C3+C4+C5+C6+C7(\lg(n))+C7+C8(\lg(n))(O(1))+C9(\lg(n))(O(1))+C10$$

$$\underbrace{C1+C2+C3+C4+C5+C6+C7+C10}_{A}+\underbrace{C7(\lg(n))+C8(\lg(n))+C9(\lg(n))}_{B(\lg(n))}$$

$$T(n) = A + B(\lg(n))$$
$$T(n) = O(1) + O(\lg(n))$$
$$T(n) = O(\lg(n))$$

5. Para esta funcion, las primeras lineas son constantes, y en el caso de la cabecera del if, se utiliza un MAX, como se asume el peor de los casos, entonces se asume que el if siempre entra, por lo que se multiplica por 1, y siempre queda un término constante, llegando a la línea del while, la cabecera se asume que entra lg(n) veces, debido a que se asume el peor de los casos, lo que hace el while es ir intercambiando elementos del arbol, haciendo que los hijos intercambien con los padres, por lo que se asume que el peor caso sería que este recorriera todo el arbol, y como la altura de un arbol se puede determinar con lg(n), se asume que el while itera lg(n)+1 veces y su cuerpo lg(n) veces

   Al realizar los calculos se obtiene que el tiempo de complejidad de la funcion insertar es de O(lg(n))

```
void minHeapify(minHeap* heap, int i) {
    int smallest = i;                                                              C1
    int l = left(i);                                                               C2
    int r = right(i);                                                              C3

    if (l < heap->size && heap->arr[l].salary < heap->arr[smallest].salary) {      C4*max(0, 1)
        smallest = l;                                                              C5
    }

    if (r < heap->size && heap->arr[r].salary < heap->arr[smallest].salary) {      C6*max(0, 1)
        smallest = r;                                                              C7
    }

    if (smallest != i) {                                                           C8*max(0, 1)
        swap(&heap->arr[i], &heap->arr[smallest]);                                 C9*O(1)
        minHeapify(heap, smallest);                                                C10*O(lg(n))
    }

}
```

C1+C2+C3+C4*max(0, 1)+C5+C6*max(0, 1)+C7+C8*max(0, 1)+C9*O(1)+C10*O(lg(n))

$$\underbrace{C1+C2+C3+C4+C5+C6+C7+C8+C9}_{A} + C10\underbrace{(lg(n))}_{B(lg(n))}$$

T(n) = A + B(lg(n))
T(n) = O(1) + O(lg(n))
T(n) = O(lg(n))

6. Para el análisis de la funcion minHeapiFy, al ser una funcion que es recursiva, se hace un análisis de todo lo que se encuentra dentro de la funcion, exceptuando la linea donde se hacen las llamadas recursivas, al analizar todo exceptuando la recursividad, se tiene que el tiempo de complejidad es O(1).

   Teniendo ya el análisis anterior, ahora se continúa con la parte recursiva, primero se debe encontrar una funcion recursica que represente a la funcion recursiva:

   Debemos determinar cuántos nodos se encuentran en cada parte del arbol, como se asume el peor de los casos, tenemos que hacer el analisis de cuantos monticulos tiene el arbol de un lado, en este cado de lado izquierdo, un arbol desbalanceado seria el peor de los casos, ya que al tener mas monticulos de un solo lado del arbol, altera demasiado el análisis, para determinar cuántos nodos se envían de lado izquierdo, tenemos que ver la tendencia, mientras más grande se va haciendo nuestro arbol, cada vez aumenta la cantidad de nodos que se encuentran de lado izquierdo, llegando a un punto en el que se observa, que converge a aproximadamente 2/3 o el 66% de los nodos de todo el arbol, por lo que podemos decir, que la cantidad de nodos, que se encuentran del lado izquierdo del arbol, es de 2/3.

Encontrando ya la cantidad de nodos que posee el arbol izquierdo, con ayuda del análisis anterior que se hizo del resto de la funcion, podemos construir la funcion de recurrencia, quedando la recurrencia:

$T(2n/3) + O(1)$

Teniendo ya la funcion de recurrencia, ahora podemos resolver, para encontrar el tiempo de complejidad de la funcion recursiva.

Utilizando el teorema maestro, primero evaluamos si se puede utilizar, verificando si cumple con lo siguiente:

$aT(n/b) + O(n^d))$; $a > 0$, $b > 1$ y $d >= 0$

En este caso cumple, ya que tenemos $a = 1$, $b = 3/2 = 1.5$ y $d = 0$

Aplicando teorema maestro, obtenemos que:

$\log_b(a) = \log_{3/2}(1) = 0$
$d = 0$
$d = \log_b(a)$

por lo tanto:
El orden de magnitud de la recursividad es:
$O(n^d lg(n)) = O(n^0 lg(n)) = O(lg(n))$

Sabiendo el tiempo de complejidad de la línea donde se hace la recursividad, podemos ya determinar el orden de magnitud de la funcion, obteniendo que el orden de magnitud es $O(lg(n))$

```
void buildMinHeap(minHeap* heap) {
    for (int i = (heap->size / 2) - 1; i >= 0; i--) {    C1(n+1)
        minHeapify(heap, i);                             C2(n)(lg(n))
    }                                                    C3
}
```

$$C1(n+1)+C2(n)(O(lg(n)))+C3$$

$$C1(n)+C1+C2(n)(lg(n))+C3$$

$$\underbrace{C1+C3}_{A}+\underbrace{C1(n)}_{B(n)}+\underbrace{C2(n)(lg(n))}_{C(n)(lg(n))}$$

$$T(n) = A + B(n) + C(n)(lg(n))$$

$$T(n) = O(1) + O(n) + O(nlg(n))$$

$$T(n) = O(nlg(n))$$

7. Para analizar esta funcion, tenemos que analizar tambien la funcion que esta mandando a llamar, en este caso minHeapiFy ya la hemos analizado anteroriormente, por lo que el tiempo de complejidad de esa línea es O(lg(n)), en el caso del for, se observa que inicia con la mitad del tamaño del arbol, y luego va decrementando hasta llegar a 0, por lo que podemos asumir que el tiempo de complejidad del cuerpo del for es lineal (n)

   Al tener ya el tiempo de complejidad hacemos los calculos de cada linea, y obtenemos que el tiempo de complejidad de la funcion buildMinHeap es O(nlg(n))

```
void print_heap(minHeap* heap) {
    cout << "-------------------------------------" << endl;            C1
    cout << "|              Empleados            |" << endl;            C2
    cout << "-------------------------------------" << endl;            C3
    cout << "|      Nombre       |       Salario     |" << endl;        C4
    cout << "-------------------------------------" << endl;            C5
    for (int i = 0; i < heap->size; i++) {                              C6(n+1)
        cout << heap->arr[i].name << "      " << heap->arr[i].salary << endl;   C7(n)
    }                                                                   C8
}
```

C1+C2+C3+C4+C5+C6(n+1)+C7(n)+C8

C1+C2+C3+C4+C5+C6(n)+C6+C7(n)+C8

$$C1+C2+\underbrace{C3+C4+C5}_{A}+C6+C8+\underbrace{C6(n)+C7(n)}_{B(n)}$$

T(n) = A + B(n)

T(n) = O(1) + O(n)

T(n) = O(n)

8. La funcion print_heap, en las primeras líneas, se obtiene que son constantes, llegando al bucle for, este tiene una complejidad lineal, ya que inicializa en 0 y de ahi va aumentando hasta ser igual que el tamaño del arbol, con esto se concluye que la funcion print_heap tiene un orden de magnitud O(n)

```c
void heapSort(minHeap* heap) {
    buildMinHeap(heap);                                  C1*O(nlg(n))
    int originalSize = heap->size;                       C2
    for (int i = heap->size - 1; i > 0; i--) {           C3(n+1)
        swap(&heap->arr[0], &heap->arr[i]);              C4*(n)O(1)
        heap->size--;                                    C5(n)
        minHeapify(heap, 0);                             C6(n)*O(lg(n))
    }
    heap->size = originalSize;                           C7
}
```

$$C1*O(nlg(n))+C2+C3(n+1)+C4(n)(O(1))+C5(n)+C6(n)(O(lg(n)))+C7$$

$$C1(nlg(n))+C2+C3(n)+C3+C4(n)+C5(n)+C6(n)(lg(n))+C7$$

$$\underbrace{C2+C3+C7}_{A}+\underbrace{C3(n)+C4(n)+C5(n)}_{B(n)}+\underbrace{C1(nlg(n))+C6(nlg(n))}_{C(nlg(n))}$$

$$T(n) = A + B(n) + C(nlg(n))$$
$$T(n) = O(1) + O(n) + O(nlg(n))$$
$$T(n) = O(nlg(n))$$

9. Para el análisis de heapSort, se analizó primero las funciones de buildMinHeap y minHeapiFy, obteniendo que su tiempo de complejidad es O(nlg(n)) para buildMinHeap y O(lg(n)) para minHeapiFy, con esto solo nos resta analizar lo demas dentro de la funcion, y realizamos los calculos, llegando a la conclusion que el tiempo de complejidad de heapSort es O(nlg(n)), al ser los lg(n)) que predominan, al ser un caso peor que los lineales que nos da el bucle for

main.cpp > main()

```cpp
#include <iostream>
#include "employee.h"
#include "minHeap.h"

using namespace std;

```

O(1)

10. En el archivo main, el inicio solo es llamado a las librerias donde se encuentran

```cpp
int main() {

    Employee* employees = load_data();          C1*O(n)
    int size = 1000;                            C2
    minHeap* heap = init_MinHeap(size);         C3*O(1)

    for (int i = 0; i < size; i++) {            C4(n+1)
        heap = insert(heap, employees[i]);      C5(n)*O(lg(n))
    }

    heapSort(heap);                             C6*O(nlg(n))
    print_heap(heap);                           C7*O(n)

    return 0;                                   C8
}
```

C1*O(n)+C2+C3*O(1)+C4(n+1)+C5(n)*O(lg(n))+C6*O(nlg(n))+C7*O(n)+C8

C1(n)+C2+C3+C4(n)+C4+C5(n)+C6(nlg(n))+C7(n)+C8

$$\underbrace{C2+C3+C4+C8}_{A}+\underbrace{C1(n)+C4(n)+C7(n)}_{B(n)}+\underbrace{C5(nlg(n))+C6(nlg(n))}_{C(nlg(n))}$$

T(n) = A + B(n) + C(nlg(n))

T(n) = O(1) + O(n) + O(nlg(n))

T(n) = O(nlg(n))

11. En el main, comienza llamando a la funcion load_data, como ya se a analizado su tiempo de complejidad, sabemos que esa linea tiene un tiempo O(n), luego se encuentra que llama a la funcion init_MinHeap, que de igual manera, ya fue analizada, lo que implica que el orden de magnitud de esa linea es O(1), en las lineas donde se encuentra el bucle for, encontramos que el orden de magnitud de su cabecera es lineal, y el el de su cuerpo es O(nlg(n)), esto porque solo el cuerpo del for se ejecuta n veces, pero dentro de el se encuentra un llamado a la funcion insert, la

cual ya se le a realizado el analisis y obtuvimos que su tiempo de complejidad es O(lg(n)), ya que el for es lineal, se multiplica por el orden de magnitud de la funcion insert, lo cual da hace que el tiempo de complejidad de esa linea sea O(nlg(n)), luego se hace llamado a heapSort, el cual obtuvimos un orden de magnitud de O(nlg(n)) y por ultimo un llamado a la funcion print_heap, con orden de magnitud O(n), haciendo todo el analisis formal del main, tenemos que el tiempo de complejidad de nuestro programa es O(nlg(n)), el cual es un caso peor que uno lineal, pero mucho mejor que uno cuadratico, por lo que se concluye que el programa es eficiente gracias al uso de los monticulos.